Concurrent Program Verification Using Likely Dataflow Invariants and Proof-guided Refinement

Outline

- Background
- Problem
- Motivation
- Generating likely data-flow Invariants
- Underapproximation
- Our Idea
- Technical Details

Background

- Safety property verification (assertion checking) on asynchronous concurrent programs with shared memory.
- Is property φ safe in $L_C \cap L_D$ where L_C and L_D are control and data flow of the program.
- Harder than sequential programs (with procedures undecidable[Ramalingam]).
- However, with restriction on L_D we can transform concurrent program to non-deterministic sequential program.

Background

Restriction on L_D

- Context Bounded Analysis
  - Transformation to context bounded sequential programs.
  - First proposed in [Qadeer and Wu] and generalized in [Lal and Reps].
- Memory Unwinding
  - Bound on number of writes to shared memory.
  - Source to source transformation. [Tomasco et al.] and [Anand et al.]
- Partial orders with loop unwinding BMC [Alglave et al.].

Background

Sequentialization using Timestamps

```
int i = 1, j = 1;
void f1() {
i = i + j;
i = i + j;
}
void f2() {
j = j + i;
j = j + i;
}
void main() {
thread t1, t2;
create(t1, f1);
create(t2, f2);
join t1;
join t2;
assert(j <= 7 );
}
```

Original Program

pos	thr	var	val
1	1	i	2
2	2	j	3
3	2	j	5
4	1	i	7

Timestore

```
int i = 1, j = 1, ct;
void f1() {
write_i(read_i() + read_j());
write_i(read_i() + read_j());
}
void f2() {
write_i(read_j() + read_i());
write_i(read_j() + read_i());
}
void main() {
f1();
f2();
assert(j <= 7 );
}
```

Transformed Program

Background

Sequentialization using Timestamps

```
void write_i(int value) {
  unsigned short loc=*;
  assume(loc_i < loc < MAXW_i &&
             free_i[loc] &&
             ts_i[loc] > ct &&
             value_i[loc] == value);
  loc_i = loc;
  free_i[loc] = false;
  icount++;
  ct = ts_i[loc];
}
```

```
int read_i() {
  unsigned short loc=*;
  assume(loc_i <= loc < MAXW_i &&
             ts_i[loc+1] > ct);
  loc_i = loc;
  if (ct<ts_i[loc]) ct=ts_i[loc];
  return value_i[loc];
}
```

Problem

- Sequentialization [Anand et al.]: Is property φ safe in $L_C \cap L_D$
  - L_C is control flow of individual threads composed sequentially.
  - L_D is all possible data-flow of shared memory represented using timestamps constraints.
- As the number of writes to consider increases so does the search space of the problem.
- Do we have to consider every possible data-flow?

Motivating Example 1

Unsafe Program

```
x=0, y=0

T1
1    while (y < NUM) {
//y = y + 1
2       t = read_y();
3       write_y(t+1);
//x = x + y
4       t1 = read_x();
5       t2 = read_y();
6       write_x(t1+t2);
7    }

T2

1      while (y < NUM) {}
2      t = read_y();
// y = x + y
3      t1 = read_x();
4      t2 = read_y();
5      write_y(t1+t2)
6      assert(y == x + t)
```

When y's value is NUM-1 following sequence can violate assert():
T1_2 → T1_3 → T2_1 → T2_2 → T2_3 → T2_4→ T1_4 → T1_5 → T1_6 → T2_5 → T2_6

To infer this sequentialization considers both local(T1_3) and remote(T2_5) writes to read of y in T1_2 and T1_5.

However by considering only local writes to y we can still find this failure.

Motivating Example 1

Safe Program

```
x=0, y=0

T1
1    while (y < NUM) {
//x = x + y
2        t1 = read_x();
3        t2 = read_y();
/y = y + 1
4        write_x(t1+t2);
5        t = read_y();
6        write_y(t+1);
7    }

T2

1     while (y < NUM) {}
2     t = read_y();
//y = x + y
3     t1 = read_x();
4     t2 = read_y();
5     write_y(t1+t2)
6     assert(y == x + t)
```

Read of y at T1_3 and T1_5 always reads local write T1_6.

But sequentialization considers both local(T1_6) and remote(T2_5) writes.

Observations

- Using $L_C \cap L_{D'}$ where $L_{D'} \subset L_D$ can reduce the search space.
  - Prove safety of φ is irrelevant of $L_{D'} \subset L_D$
  - Or prove $L_D \setminus L_{D'}$ is infeasible.
  - Or widen $L_{D'}$ towards L_D
- We can use UNSAT proof to prove irrelevance and widen $L_{D'}$ towards L_D
- Data-flow invariants can provide $L_{D'}$ where $L_{D'} \subseteq L_D$
  - Dynamic analysis tools can generate likely invariants faster.

DefUse Invariants

- DefUse[Shi et al.] is a dynamic analysis based bug-detection tool.
- Extracts definition and use relationship ("developer's intended data-flow") by running the program under its test suite.
- Violation of discovered invariants are considered as potential bugs.
- Can generate false positives.

DefUse Invariants

- Local/Remote: Read either reads from local thread or from remote thread.

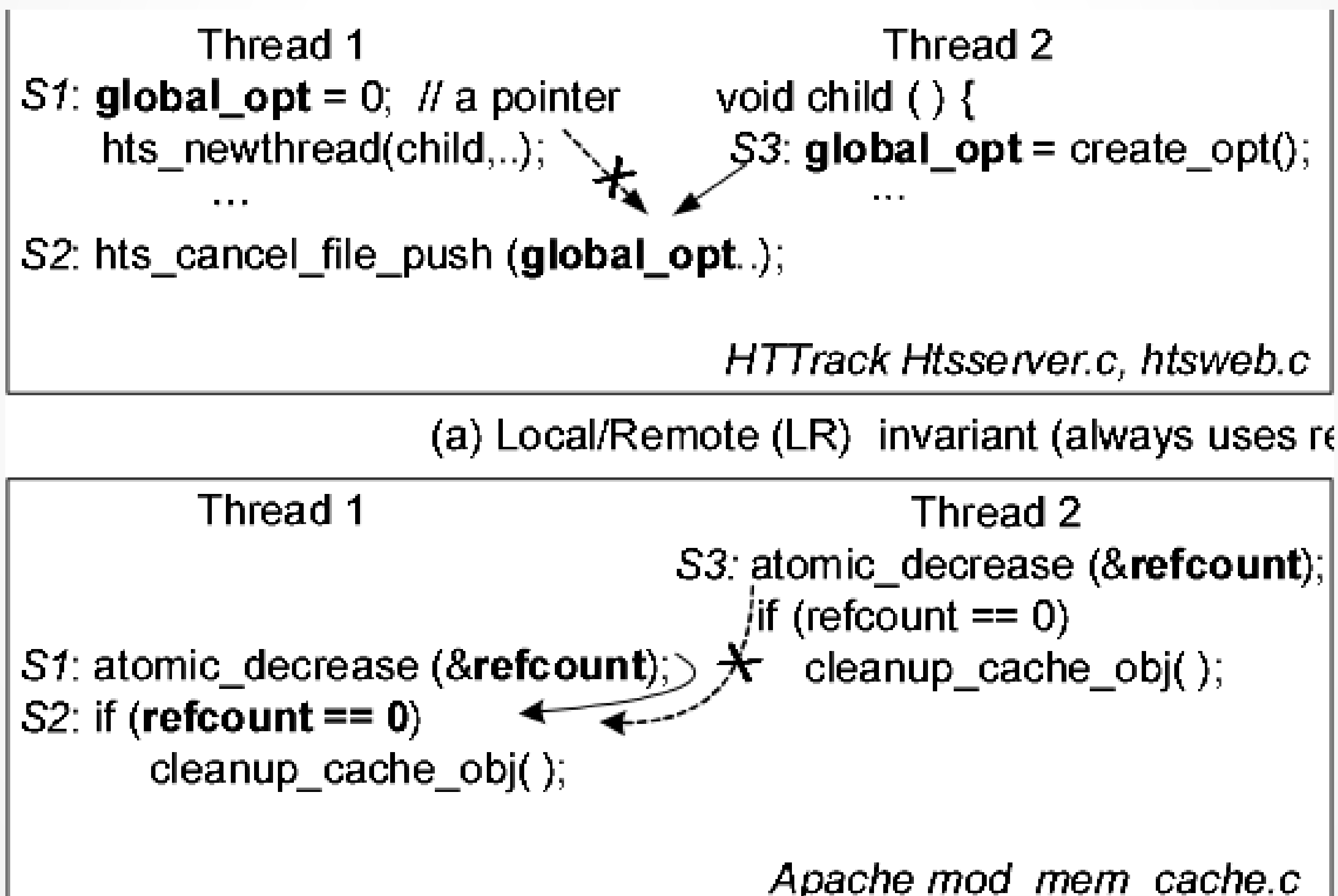


Image Reference: [Shi et al.]

DefUse Invariants

- Follower: Two consecutive reads uses same write.

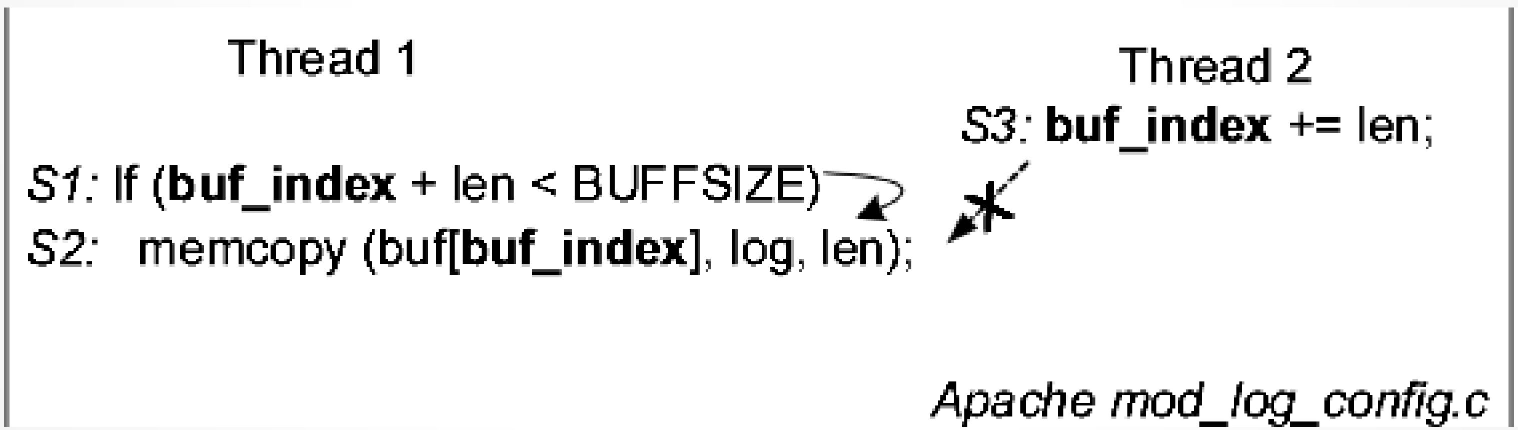


Image Reference: [Shi et al.]

DefUse Invariants

- Definition Set: A read reads only from a set of writes.

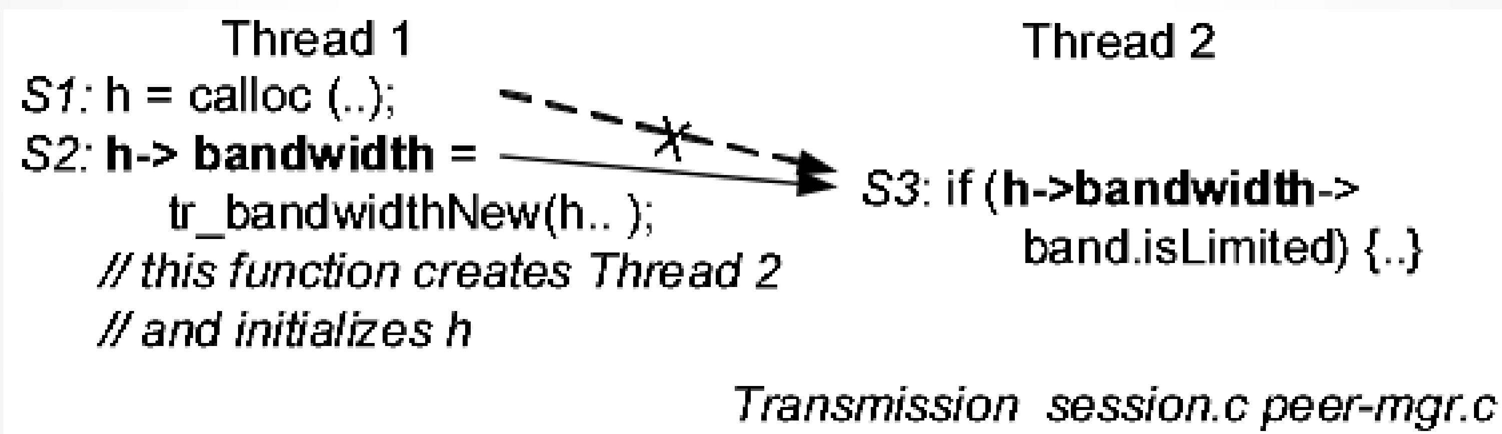


Image Reference: [Shi et al.]

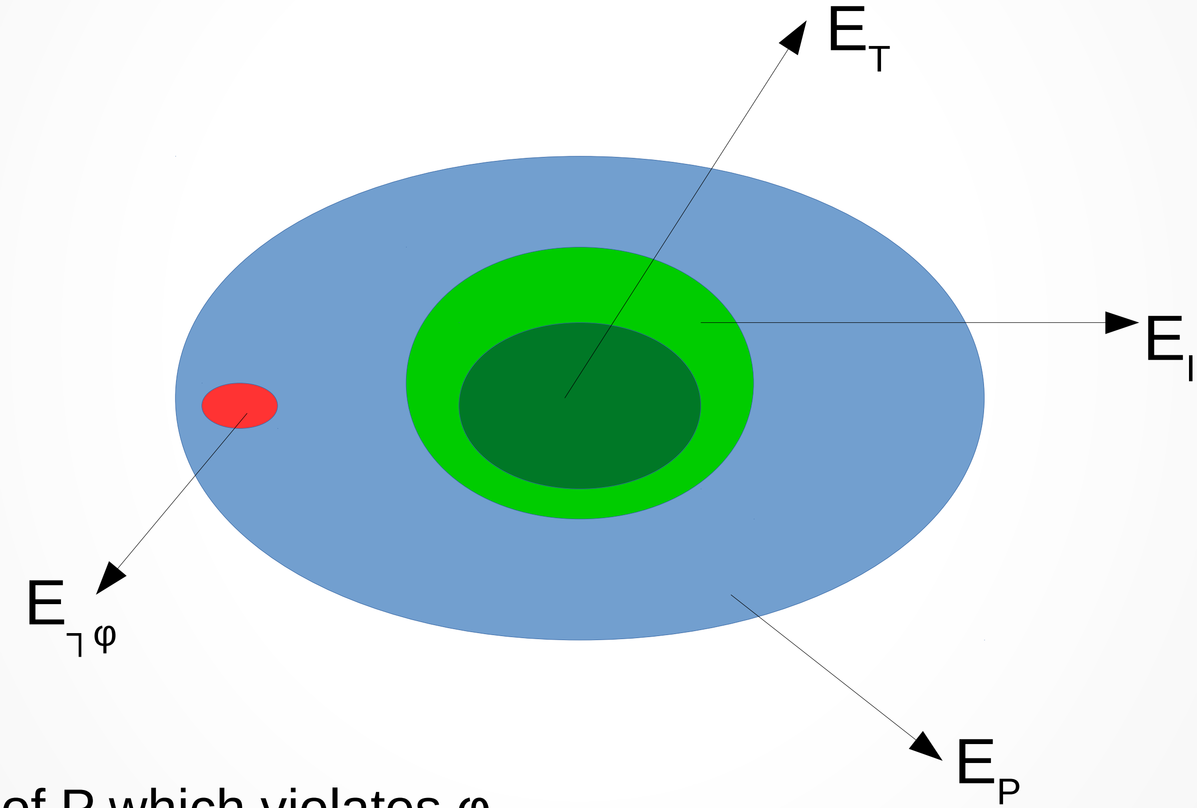
Proof by Underapproximation

- $P' := P \wedge (I_1 \wedge I_2 \wedge \ldots \wedge I_n)$ is an underapproximation of P, where $I_1 \ldots I_n$ are referred as constraints.
  - P', P and $I_1 \ldots I_n$ are in CNF.
- If P' is satisfiable then so is P.
- UNSAT core/proof of P' is a subset of clauses of P' that is unsatisfiable.
- Given an UNSAT core of P' if none of $I_1 \ldots I_n$ are present in it then P is unsatisfiable.

Our Idea

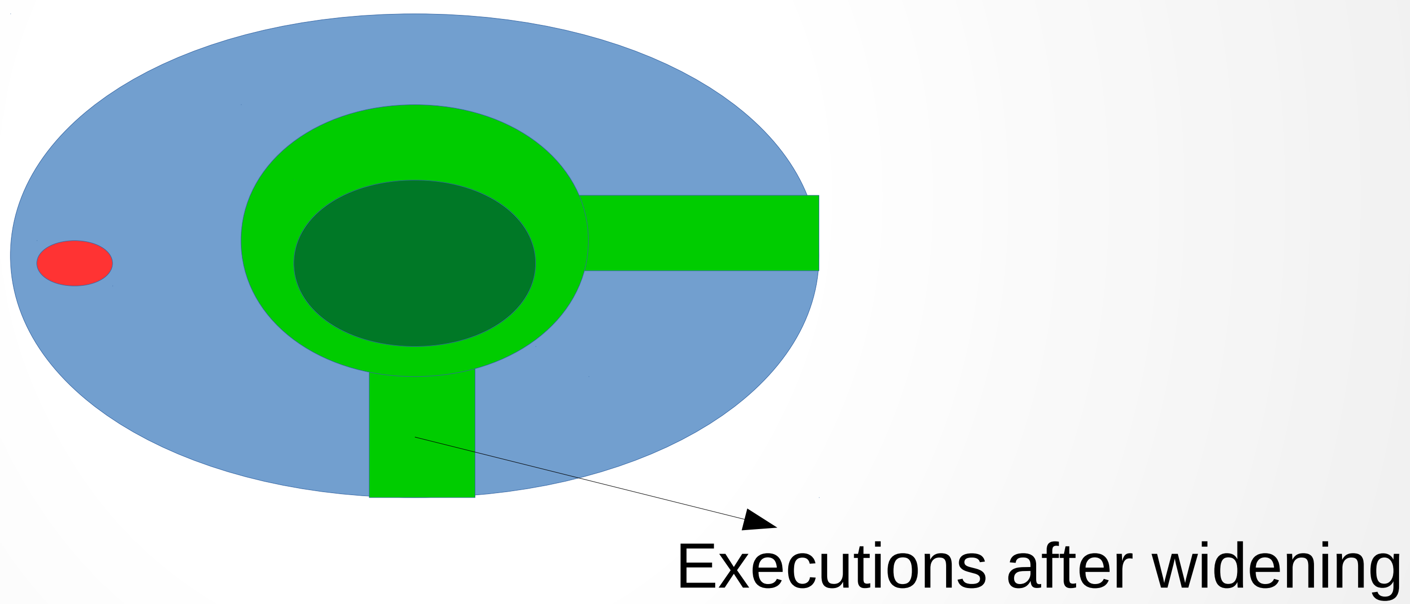
- We conjunct constraints to $L_C \cap L_D$ to get $L_C \cap L_{D'}$ where $L_{D'} \subseteq L_D$
- This restriction gives us an underapproximation of original program.
- If the property is violated in this model then we report counterexample.
- Otherwise we refine/widen $L_{D'}$ using UNSAT core.

Our Idea

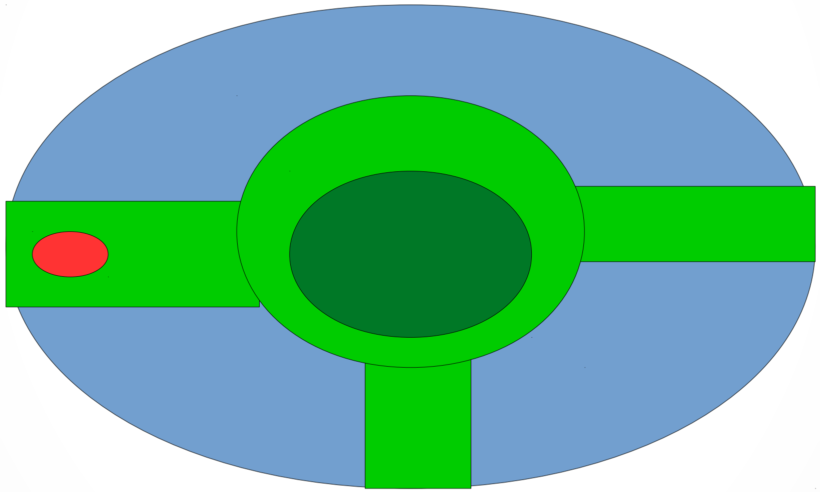


$E_{\neg\varphi}$ Executions of P which violates φ
E_T Executions under invariant generation tool
E_I Executions that are covered by inavriants
E_P Actual program Executions

Our Idea

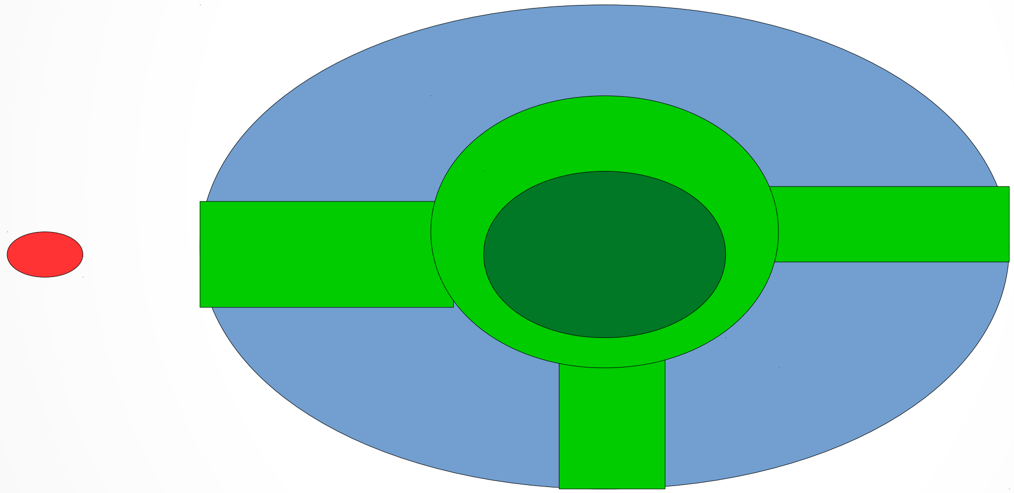


Our Idea



UNSAFE

Our Idea



SAFE

Executions are irrelevant to the property or infeasible

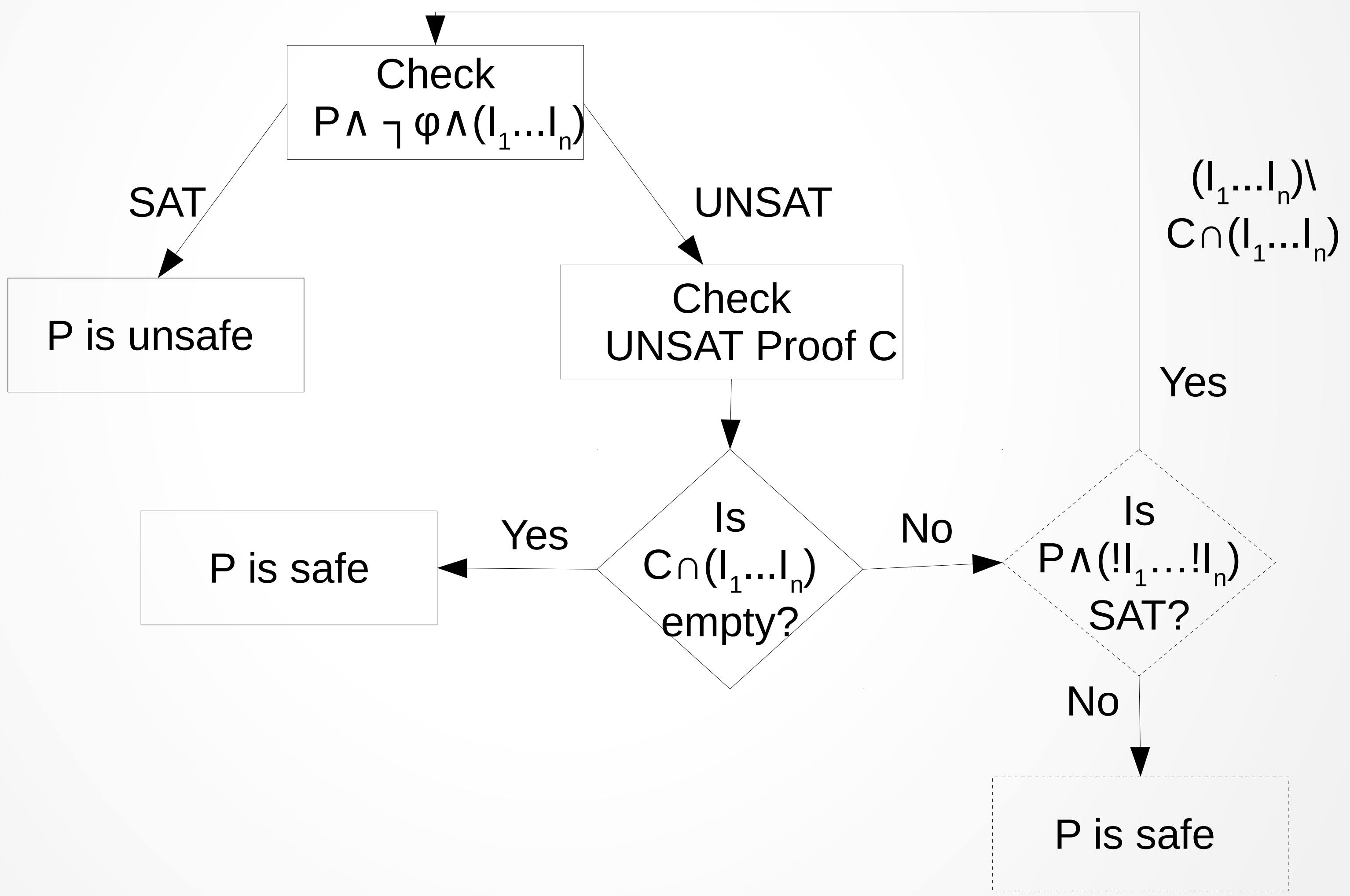
Our Idea

Different Strategies for Constraints

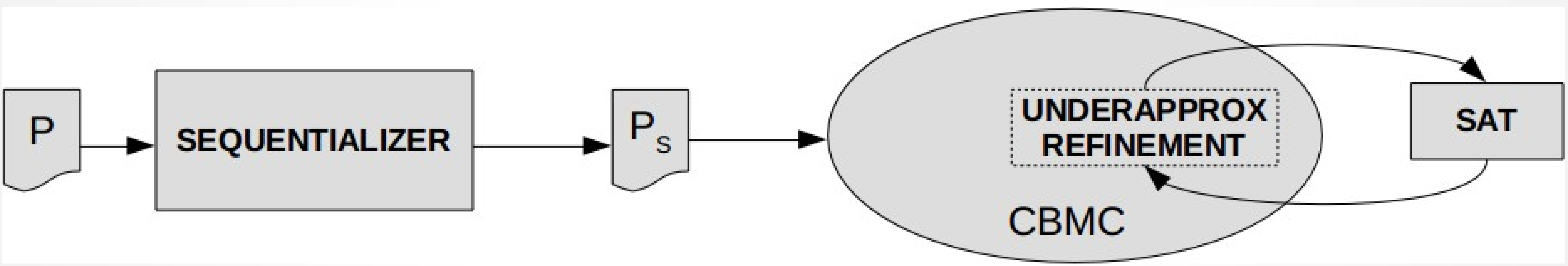
- Restrict each read to consider only writes from likely data-flow invariants.
  - In time store constrain read to map write set deduced by likely invariants.
- Starting from a subset of writes increase number of writes(monotonically increasing sets) to consider for each read.
  - If 1 to n positions of time store are allowed for a read then start with 1 to j, where j < n, and increment j towards n.
- [Grumberg et al.] considers number of interleavings for BMC of asynchronous concurrent system.

Our Idea

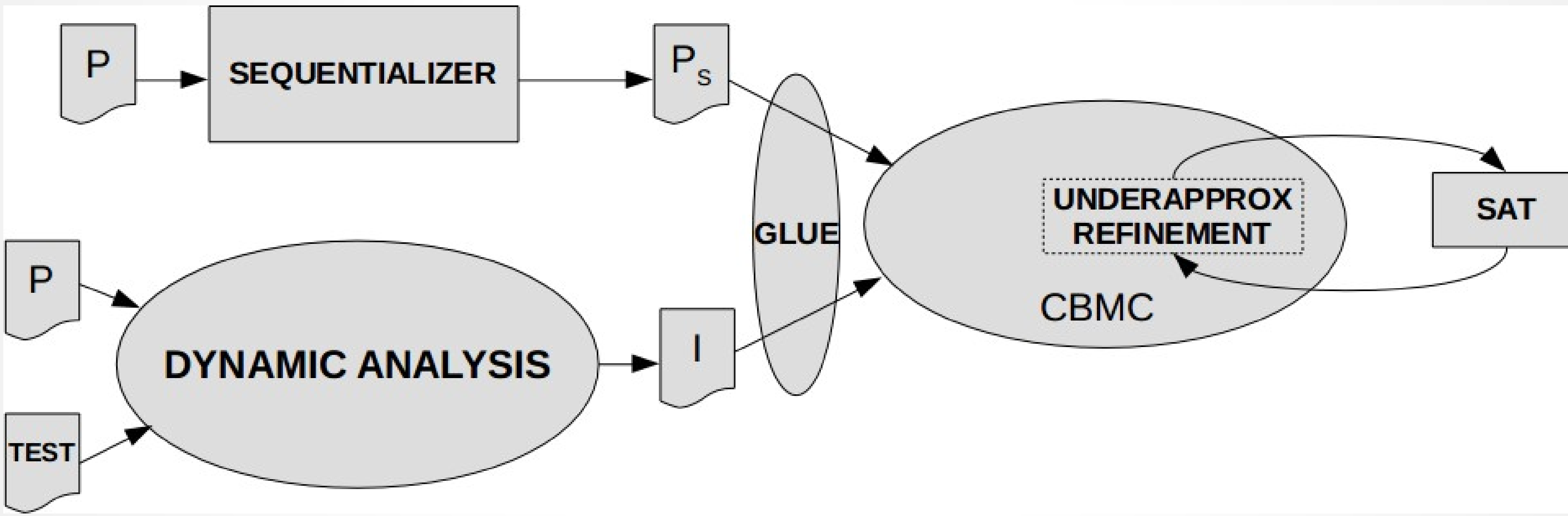
Strategies for Refinement



Design



Design



P
SEQUENTIALIZER
P S
GLUE
UNDERAPPROX
REFINEMENT
SAT
CBMC
P
DYNAMIC ANALYSIS
I
TEST

Technical Details

- Goals:
  - Underapproximation-refinement loop.
  - Encoding Defuse constraints in sequentialization.
  - Discovering DefUse invariants. (In Progress)
  - Experimentation. (In Progress)

Technical Details

```
int read_i() {
 unsigned short loc;
 assume(iloc <= loc < MAXW);
 assume(ts_i[loc+1] > ct);
 iloc = loc;
 if (ct<ts_i[loc]) ct=ts_i[loc];
 return value_i[loc];
}
```

Sequentialization from [Anand et al.]

Technical Details
Encoding

- Local/Remote
  - loc=iloc for local and loc > iloc for remote
- Follower
  - prev_loc = cur_loc
- Write Set
  - loc=w1 || loc=w2 ….
- Passed as additional constraints along with a switch variable (e12 → (loc = iloc) && !e12 → (iloc <= loc < MAXW)).
- Arbitrary write set
  - (e12 → (iloc <= loc < MAXW/2) && !e12 → (iloc <= loc < MAXW))

Technical Details
Refinement

- Implemented in CBMC Solver module.
- Uses MiniSAT's conflict feature.
- Initially all switch variables are assumed to be true.
- If none of switch variables are part of conflict then we return program is safe.
- If a switch variable is part of conflict then that variable is dropped from assumption.

Technical Details
Discovering Likely Invariants

- Defuse tool is not available.
- Currently implementing dynamic analysis tool using PIN.
- PIN can add hooks in binary which can be used to track reads.
- Requires address to symbol name translation, which is not provided by PIN.

Technical Details

Experiment

- Read can have potentially large set of writes.
- SVCOMP tests
- Benchmark used by ESMBC in [Cordeiro and Fischer]
- Benchmark used by Inspect, a runtime model checker [Yang et al.]
- Benchmarks used in systematic concurrency testing like [Wang and Hoang] and [Wang et al.]

Preliminary Experiment

With True Invariants (gain)

File	Flags	Runtime of decision Procedure	Refinement Progress
0_false_seq.c	NUM=5, unwind=10, status=VERIFICATION FAILED	2.759s	
0_false_seq_inv.c	NUM=5, unwind=10, status=VERIFICATION FAILED	2.548s	30, 1 iteration
0_false_seq.c	NUM=10, unwind=15, status=VERIFICATION FAILED	7.691s	
0_false_seq_inv.c	NUM=10, unwind=15, status=VERIFICATION FAILED	7.166s	45, 1 iteration
0_false_seq.c	NUM=15, unwind=20, status=VERIFICATION FAILED	19.602s	
0_false_seq_inv.c	NUM=15, unwind=20, status=VERIFICATION FAILED	14.439s	60, 1 iteration
0_false_seq.c	NUM=20, unwind=25, status=VERIFICATION FAILED	37.28s	
0_false_seq_inv.c	NUM=20, unwind=25, status=VERIFICATION FAILED	28.155s	75, 1 iteration

0_true_seq.c	NUM=15, unwind=20, status=VERIFICATION SUCCESSFUL	11.603s	
0_true_seq_inv.c	NUM=15, unwind=20, status=VERIFICATION SUCCESSFUL	8.647s	75, 1 iteration
2_true_seq.c	NUM=5, unwind=15, status=VERIFICATION SUCCESSFUL	0.557s	
2_true_seq_inv.c	NUM=5, unwind=15, status=VERIFICATION SUCCESSFUL	0.428s	10 to 0 in 2 iterations

Preliminary Experiment

With True Invariants (degenerates)

2_true_seq.c	NUM=10, unwind=25, status=VERIFICATION SUCCESSFUL	7.399s	
2_true_seq_inv.c	NUM=10, unwind=25, status=VERIFICATION SUCCESSFUL	8.609s	20 to 0 in 2; without property: 1.468s(UNSAT) + with only invariants: 3.917s
2_true_seq.c	NUM=15, unwind=35, status=VERIFICATION SUCCESSFUL	73.551s	
2_true_seq_inv.c	NUM=15, unwind=35, status=TIMEOUT	82.43s	30 to 0 in 2; without property: 6.698s(UNSAT) + with only invariants: 14.209s(UNSAT);
2_true_seq.c	NUM=20, unwind=45, status=TIMEOUT	365.19s	
2_true_seq_inv.c	NUM=20, unwind=45, status=TIMEOUT	889.136s	40 to 0 in 2; without property: 14.346s(UNSAT) + with only invariants: 153.443s(UNSAT)

$P \wedge (!I_1 \ldots !I_n)$ check refinement performs better

1_true_seq.c	NUM=20, unwind=45, status=VERIFICATION SUCCESSFUL	6.014s	
1_true_seq_inv.c	NUM=20, unwind=45, status=VERIFICATION SUCCESSFUL	7.129s	40 to 0 in 2 iterations; 14.676s UNSAT;
1_true_seq.c	NUM=25, unwind=55, status=VERIFICATION SUCCESSFUL	9.368s	
1_true_seq_inv.c	NUM=25, unwind=55, status=VERIFICATION SUCCESSFUL	12.439s	50 to 0 in 2; without property: 22.168s UNSAT;
1_true_seq.c	NUM=30, unwind=65, status=VERIFICATION SUCCESSFUL	14.9s	
1_true_seq_inv.c	NUM=30, unwind=65, status=VERIFICATION SUCCESSFUL	18.14s	60 to 0 in 2; without property: 33.095s UNSAT;

Preliminary Experiment

Arbitrary write set

File	Flags(Timeout was 300s.)	Runtime of decision Procedure	Assumptions
fb_false.c	NUM=5, unwind=7	0.875s	
fb_false_inv.c		0.661s	20 to 6 in 5 iterations
fb_true.c	NUM=5, unwind=7	1.415s	
fb_true_inv.c		0.794s	20 to 5 in 7 iterations
fb2_false.c	NUM=6, unwind=8	1.311s	
fb2_false_inv.c		1.616s	24 to 7 in 7 iterations
fb2_true.c	NUM=6, unwind=8	3.428s	
fb2_true_inv.c		2.644s	24 to 7 in 7 iterations
fb3_false.c	NUM=11, unwind=15	44.833s	
fb3_false_inv.c		65.757s	44 to 8 in 5 iterations
fb3_true.c	NUM=11, unwind=15	271.689s	
fb3_true_inv.c		170.751s	44 to 3 in 6 iterations

Fibonacci programs from SVCOMP 2017
4/6 programs performs better

Preliminary Experiment

Refinement Strategies

iloc <= loc < MAXW	Point of reference.
loc == iloc	When true invariant: performs better in unsafe programs. Takes more time in safe except few cases. When not true invariant: takes more time.
iloc <= loc < MAXW/2	Performs better for Fib programs.
!(loc == iloc)	Takes more time.
P ∧ !(loc == iloc)	For one program time taken to check this and verification is less.

Next steps

- Defuse like Invariant generation using PIN.
- Experiment on different benchmarks.

Future Directions

- Using likely inductive invariants to reduce L_c.
- Partitioning of traces [Farzan et al.].
- Technical
  - Finer refinement.
  - Experiment on real world software like Linux (Drivers or part of Implementation), Apache httpd, etc.
    - Require sequentialization support.

References

- [Grumberg et al.]: O. Grumberg et al. Proof-guided underapproximation-widening for multi-process systems. POPL, pp. 122–131, 2005
- [Farzan et al.]: A. Farzan, Z. Kincaid, and A. Podelski. Inductive data ow graphs. In POPL, pages 129-142, 2013.
- [Cordeiro and Fischer] L. Cordeiro and B. Fischer. Verifying multi-threaded software using SMT based context-bounded model checking. In ICSE pages 331–340. ACM, 2011.
- [Yang et al.] Yang, Y., Chen, X., Gopalakrishnan, G.: Inspect: A Runtime Model Checker for Multi-threaded C Programs. Technical Report UUCS-08-004, University of Utah (2008).
- [Wang and Hoang] Chao Wang and Kevin Hoang. Precisely deciding control state reachability in concurrent traces withlimited observability. In Proceedings of VMCAI'14, Verifcation, Model Checking, and Abstract Interpretation, pages 376{394. Springer-Verlag(LNCS), 2014.
- [Wang et al.] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In ICSE, pages 221–230, 2011.
- Project wiki: http://www.cmi.ac.in/~sumanth/dokuwiki/doku.php?id=invariants:underapproximation

References

- [Ramalingam] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. In TOPLAS, 2000
- [Qadeer and Wu] S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In PLDI, 2004
- [Lal and Reps] Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. Formal Methods in System Design 35 (2009) 73–97
- [Tomasco et al.] Tomasco, E., Inverso, O., Fischer, B., La Torre, S., Parlato, G.: Verifying concurrent programs by memory unwinding. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 9035, pp. 551–565. Springer Berlin Heidelberg (2015)
- [Anand et al.]: Sequentialization Using Timestamps, Anand Yeolekar, Kumar Madhukar, Dipali Bhutada, Venkatesh R. In TAMC, 2017.
- [Alglave et al.] Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Computer Aided Verification. pp. 141–157. Springer (2013)
- [Shi et al.]: Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. Do i use the wrong definition? defuse: Definition-use invariants for detecting concurrency and sequential bugs. In OOPSLA,2010

References

- [Grumberg et al.]: O. Grumberg et al. Proof-guided underapproximation-widening for multi-process systems. POPL, pp. 122–131, 2005
- [Farzan et al.]: A. Farzan, Z. Kincaid, and A. Podelski. Inductive data ow graphs. In POPL, pages 129-142, 2013.
- [Cordeiro and Fischer] L. Cordeiro and B. Fischer. Verifying multi-threaded software using SMT based context-bounded model checking. In ICSE pages 331–340. ACM, 2011.
- [Yang et al.] Yang, Y., Chen, X., Gopalakrishnan, G.: Inspect: A Runtime Model Checker for Multi-threaded C Programs. Technical Report UUCS-08-004, University of Utah (2008).
- [Wang and Hoang] Chao Wang and Kevin Hoang. Precisely deciding control state reachability in concurrent traces withlimited observability. In Proceedings of VMCAI'14, Verifcation, Model Checking, and Abstract Interpretation, pages 376{394. Springer-Verlag(LNCS), 2014.
- [Wang et al.] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In ICSE, pages 221–230, 2011.

- Project wiki: http://www.cmi.ac.in/~sumanth/dokuwiki/doku.php?id=invariants:underapproximation